The Journal of Supercomputing (2020) 76:8313–8323
https://doi.org/10.1007/s11227-019-03114-y

An accelerated rendering scheme for massively large point cloud data

Nakhoon Baek[1] · Kwan-Hee Yoo[2]

Published online: 18 December 2019
© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract
In the field of large-scale data visualization, the graphics rendering speed is one of the most important factors for its application development. Since the large-scale data visualization usually requires three-dimensional representations, the three-dimensional graphics libraries such as OpenGL and DirectX have been widely used. In this paper, we suggest a new way of accelerated rendering, through directly using the direct rendering manager packets. Current three-dimensional graphics features are focused on the efficiency of general purpose rendering pipelines. In contrast, we concentrated on the speed-up of the special-purpose rendering pipeline, for point cloud rendering. Our result shows that we achieved our purpose effectively.

Keywords Large-scale data visualization · Direct rendering manager · Graphics acceleration · Point rendering

1 Introduction

In the area of three-dimensional computer graphics, they focused on the both side of speed and simplicity of the visualization techniques. To show much more realistic scenes, they need precise and accurate numerical data on the graphics models. The core of the technique is how to efficiently and rapidly display those data on the screen. In contrast, they also pursue the easy and intuitive way of handling those big size data [16, 22, 29].

✉ Kwan-Hee Yoo
khyoo@cbnu.ac.kr; khyoo@chungbuk.ac.kr

Nakhoon Baek
oceancru@gmail.com; nbaek@knu.ac.kr

[1] School of Computer Science and Engineering, Kyungpook National University, Daegu 41566, Republic of Korea

[2] Department of Computer Science, Chungbuk National University, Cheongju, Chungbuk 28644, Republic of Korea

At this time, OpenGL [18, 27, 30, 33] and DirectX [21] are the most widely used *Application Programmer's Interface* (API) libraries in the computer graphics field [2, 4]. Though some graphics engines and graphics tools are available, they focus on the efficient application programmer's interfaces, rather than the execution speed. Therefore, currently, these graphics libraries are regarded as the most efficient rendering ways for the large-scale precise data [22, 29].

In the field of modern computer graphics, the fundamental output units are output primitives. One of the most widely used output primitives is the triangles. Most three-dimensional graphics scenes can be represented as a very large number of triangles. Other possible three-dimensional graphics primitives include points, line segments, quadrangles, and others. Thus, modern graphics pipelines are highly tuned for triangles. In contrast, some graphics applications need other primitives. In this paper, we focused on three-dimensional points. The three-dimensional point primitives are finally mapped as two-dimensional pixels on the computer screens.

Due to the development of efficient point sampling devices, including laser scanners and *Light Detection And Ranging* (LiDAR) devices, we often get millions or even billions of three-dimensional points, as the result of point sampling processes [25, 35]. In the typical LiDAR systems, laser lights illuminate the target objects, to measure the distance from the laser scanner to the target objects. We can use the LiDAR system to obtain various kinds of geological data, including terrain surfaces and interior obstacles. Many geometric applications use the LiDAR system to build terrains, outside buildings, and geometric models [7, 9, 24, 31].

Typical LiDAR systems produce very large-scale data sets, with a large number of 3D sampling points. Due to the extremely large number of points, the laser-scanned data points are often called as *point clouds* [5, 32]. To represent these point clouds on the computer screens directly, we need a three-dimensional graphics pipeline highly tuned to those point clouds. With the development of the modern point sampling devices, we now frequently get the need to efficiently draw those point clouds.

In this paper, we will show a special-purpose graphics rendering scheme, highly optimized for the three-dimensional point clouds. We focused on the low-level data packets between the main board and the graphics cards. We designed a small-scale rendering library. We achieved this small-scale graphics rendering library, based on the *Direct Rendering Manager* (DRM) packets and application library functions. Our focus is to make a light-weight accelerated point rendering library, without graphical windowing system support, especially for small-size and/or embedded systems.

In the next section, previous works are presented. In Sect. 3, we will present the underlying techniques, including the DRM packets, graphics pipelines, shading language features, and others. Implementation results are followed. Conclusion is in the final section.

2 Previous works

Light Detection and Ranging (LiDAR) system generates a set of points to express its own 3D topographic information. LiDAR data can be displayed either through direct rendering of the point cloud or by extracting features through classification and/or segmentation [5, 32]. In any case, a potential problem is that LiDAR data sets are massively large for even small target objects.

Levoy [20] was the first one to consider points as rendering primitives for solid objects. Several tree-based data structures have been proposed. It is important that the data structure supports *levels of detail* (LOD) so that the point cloud can be rendered with a different amount of details, depending on the distance to the camera.

Gobbetti and Marton [12] proposed a rendering system called *Layered Point Clouds* (LPC). The point cloud is stored in a binary tree. By rendering just the points in the first levels of the hierarchy, a coarse approximation can be rendered.

Wimmer and Scheiblauer [37] introduced the so-called the *nested octree* as a data structure for point clouds. The nested octree is an octree whose nodes contain subsamples of the points inside the bounding box represented by the octree node, similar to LPC.

Our focus is rendering the massively large point cloud, as is, without any conversion to internal data structures. In this case, our concern is actually the drawing tools. In the history of modern computer graphics, there have been many kinds of graphics libraries, including OpenGL [18, 30], DirectX [21], X window systems [23, 38], Display PostScript [34], Cairo [8], OpenInventor [36], Qt [19], and so on. Currently, three-dimensional graphics libraries are the main stream in the computer graphics and its related areas. Most of the 3D graphics application programs use 3D graphics libraries and/or 3D graphics engines, which are based on 3D graphics libraries such as OpenGL [18, 30] and DirectX [21].

Traditionally, the three graphics pipeline adopt the *normalized device coordinate* as its reference frame for the intermediate results. For simplicity and efficiency, the normalized device coordinates (x_d, y_d, z_d) are ranged in a unit cube of $[-1, +1] \times [-1, +1] \times [-1, +1]$ in the 3D coordinate system [1, 3, 14].

For precise *texture mapping* and/or *level-of-detail* operations, we need to calculate the detail level of graphics primitives. For an edge, its edge length s can be calculated as follows:

$$s = \frac{r}{f} \cos^{-1}\left(\frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{|\mathbf{v}_1||\mathbf{v}_2|}\right),$$

where r is the specified resolution in pixels, f is the field of view, and $\mathbf{v}_1$ and $\mathbf{v}_2$ are two vectors from the camera position to the vertices of the edge. According to the magnitude of s, we can choose the suitable level-of-detail factors [26].

In some cases, we also need image-enhancement filtering techniques even for the point clouds. As an example, typical *Monte Carlo convolution* can be calculated as follows:

$$(f * g)(x) = \frac{1}{|N(x)|} \sum_{j \in N(x)} \frac{f(y_j) g\left(\frac{x - y_j}{r}\right)}{p(y_j|x)},$$

where $N(x)$ is the set of *neighborhood indices* in a sphere of radius r, and $p(\cdot)$ is the *probability density function* (PDF) [15].

3 Design of the rendering system

The initial start point of our work is the avoidance of using the graphical window systems. Modern graphical window systems have many overheads to handle the windows. Every graphical window should handle the user interactions and window-to-window events, and much more system-dependent user interface issues. In contrast, some computer graphics architectures adopt direct rendering systems, which accesses the framebuffer directly, as shown in Fig. 1. In some resource-restricted systems and/or embedded systems, the graphical windowing systems are not necessary or even should be avoided due to the limited resources.

3.1 Direct rendering manager

In the case of Linux and its derived systems, the *direct rendering manager* (DRM) module can access the framebuffer directly [11]. In modern computer graphics architecture, the *graphics-processing unit* (GPU) is essential to the framebuffer management and various graphics processing. Thus, the modern DRM modules now also manage the GPUs in addition to the traditional framebuffers.

The DRM is actually a module of the Linux kernel. It provides an *application programmer's interface* (API) to the GPU. The upper layers, including OpenGL and other application-level graphics libraries, use this DRM module as the standard way of transferring the data to the GPU. A programmer can send the rendering commands (or more explicitly, GPU machine instructions) and the target data to the GPU, through directly calling DRM functions, as shown in Fig. 2.

The DRM module provides additional functionalities including framebuffer managing, mode setting, memory-sharing objects handling, memory synchronization, and others. Some of these expansions had carried out their own specific names, such as *Graphics Execution Manager* (GEM) [28] or *Kernel Mode Setting* (KMS) [10]. Those parts are actually the sub-modules of the whole DRM module. The detailed descriptions on these modules are followed in the subsections.

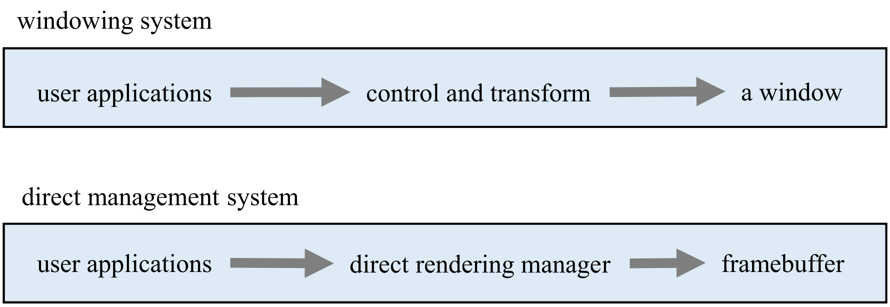

Fig. 1 Avoiding the use of graphical window systems

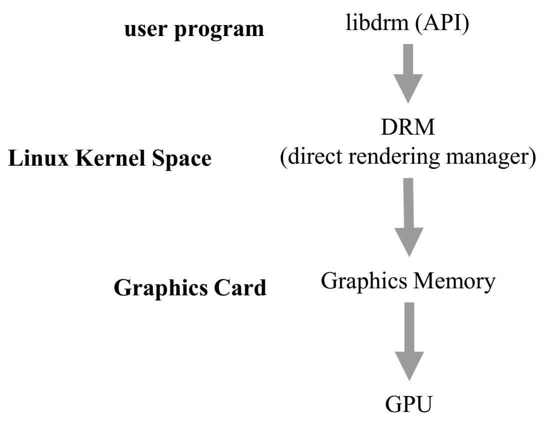

Fig. 2 The DRM module in the Linux Kernel

3.2 Graphics execution manager

To control graphics contexts and its related graphics memory areas, Linux kernels also provide another module, named the *Graphics Execution Manager* (GEM). This module provides more optimized ways to share the low-level GPU buffers and the GPU-specific contexts. The GEM was designed to manage the graphics memory areas, including the graphics image areas and texture areas [10].

Graphics data can consume arbitrary amounts of memory, with 3D applications constructing even larger sets of textures and vertices. Historically, the traditional graphics application programs send the rendering data from the main memory to the graphics memory, for each *context switching*. For more speed-ups, ensuring that graphics data remain persistent across context switches allows applications significant new functionality while also improving performance for existing API's.

Modern Linux desktops include significant 3D rendering as a fundamental component of the desktop image construction process. The 2D and 3D applications render their contents to off-screen memory areas, and the final screen image was displayed from those contents.

3.3 Kernel mode setting

Modern commercial GPUs have many selectable internal features, which can be controlled through the mode-setting commands. Those *mode-setting* commands include the setting of the screen resolutions, color bit resolutions, depth bit representations, stencil bit settings, refresh rates, and much more. These commands are transferred to the GPU before the start of the target graphics application program.

A special Linux kernel module, named *kernel mode setting* (KMS), is used to provide those mode-setting commands [10, 17]. Currently, the kernel-level implementation of KMS enables us to select the screen resolutions and the console mode switching operations.

Another important resource related to the GPU is the graphics memory. In the typical graphics execution environment, there will be several 3D graphics application programs, even with different settings for each of them. The different settings and its related graphics memory areas are referred as *graphics context* [27, 30].

4 Implementation and its results

In the case of Linux kernels, the DRM module is used to access the GPU. The upper layers, including OpenGL and other application-level graphics libraries, use this DRM module as the standard way of transferring the data to the GPU.

Typical graphics programs send the data as a mixture of the target data and the rendering commands for those data. In the case of large-scale data visualization, the portion of the target data is dramatically high, with very small amount of the rendering commands. Currently, graphics libraries, however, use the traditional way of transferring the data and commands, as a set of mixtures.

OpenGL programs to render point clouds are typically constructed with point drawing primitives and large-size vertex buffer handling commands. The size of vertex buffer itself and/or the number of points drawn by a single primitives is restricted with internal limitations of an OpenGL implementation. In our DRM-based implementation, the internal limitations of OpenGL implementations are avoided, and only the GPU-level physical limitations are applied to the rendering instructions. Since the physical limitations in modern GPUs are set to very large values, we have actually no practical limitations, in most cases.

In our DRM-based implementation, we by-pass the high-level libraries including OpenGL and similar ones. Instead, we send the DRM packets, containing low-level GPU machine instructions, directly to the GPU. The vertex data are also managed by the DRM module. In this way, we can remove the duplicated GPU-level instructions in the rendering pipelines of the OpenGL and other high-level graphics libraries. In the case of typical Vulkan-based rendering applications, as another example, they easily meet the repeated memory transfers between the *host-visible* areas and the *device-local* areas[6, 13], while our DRM modules can avoid those repeated copies, since we directly control the data in the GPU memory.

Our implementation is based on the Linux library implementation of DRM, named *libdrm*. This system library provides easy ways of sending DRM packets. Based on the DRM packets, the fundamental rendering pipeline can be easily established, as shown in Fig. 3. To minimize the implementation costs and also the rendering costs, we used an optimized OpenGL shading language program as an element of the fixed-function graphics pipeline. We also set the depth buffer (or Z-buffer) as a programmer-selectable option of the graphics pipeline. In the case of point clouds, the texture buffer and the stencil buffer are not required, at least for our application cases. So, we omitted the texture handling features and stencil buffer support.

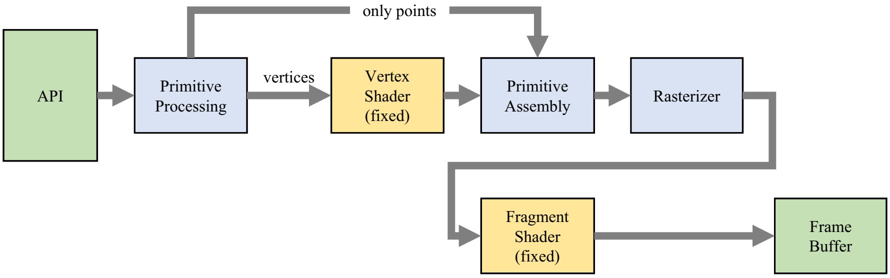

Fig. 3 Our fixed-function graphics pipeline for point cloud rendering

As a prototype implementation, we used a set of point clouds from LiDAR devices, which typically consist of more than 3 million color points, as shown in Figs. 4, 5, 6, and 7. In some OpenGL-based implementations, there may be internal limitations to the number of vertex points for each drawing commands, and also to the number of total vertex points. To solve these limitations, we split the point cloud into a set of separate rendering commands and their-related vertex data. In contrast, our DRM-based implementation can draw the whole point cloud with a single GPU-command sequences, since the physical limitations of the GPU data access size are much larger.

Table 1 shows the experimental results. In comparison with the high-level OpenGL-based rendering method with full windowing system and full rendering pipeline support, our DRM-based implementation shows 42 to 94 times accelerated rendering times. All experiments are executed on a Linux-based single board computer with Intel CPU and its embedded GPU.

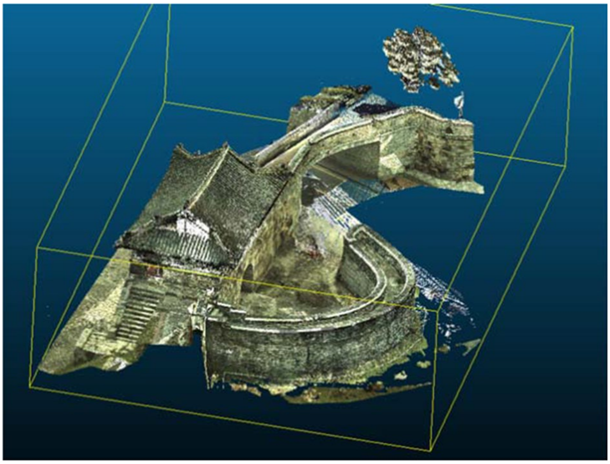

Fig. 4 An example of large-scale data visualization with our DRM-based system

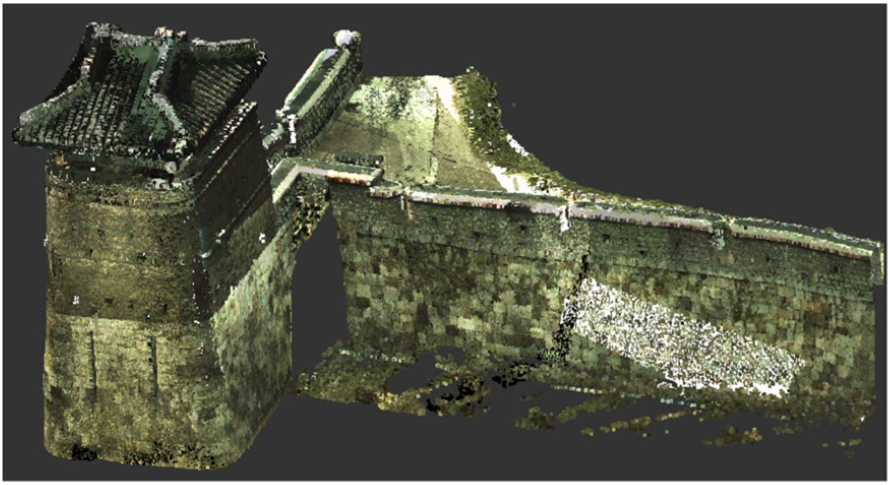

Fig. 5 Another example of large-scale data visualization from the LiDAR point cloud

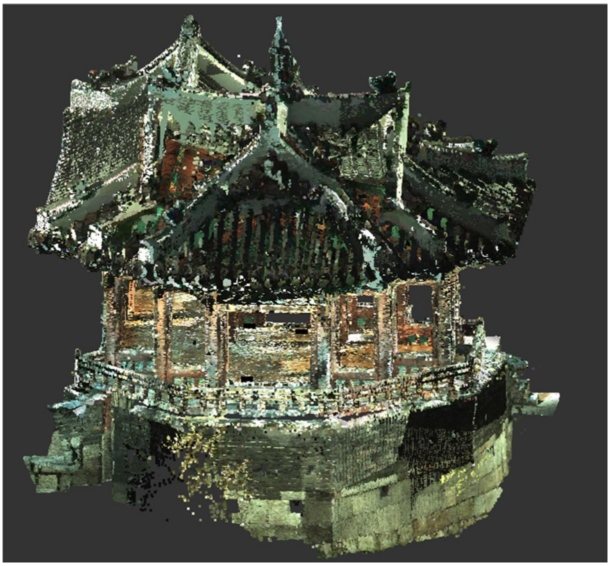

Fig. 6 An example of our LiDAR point cloud rendering system

5 Conclusion

In these days, typical graphics tools and application programming interface libraries are designed to support easy-to-use user interfaces and function calls. Currently, the graphics libraries are tuned to control the underlying graphics hardware directly, as is in the new graphics standard of *Vulkan* [13]. In the case of

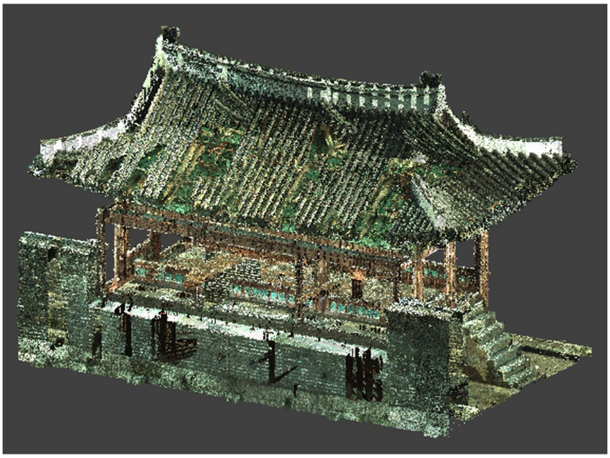

Fig. 7 Another example of our LiDAR point cloud rendering system

Table 1 Comparison of rendering times

	# of points	(a) Our DRM method (ms)	(b) Generic rendering (ms)	Ration (b/a)
Hwaseo (Fig. 4)	5,407,008	0.261	24.690	94.598
Seobuk (Fig. 5)	4,219,145	0.256	18.671	72.934
Banghwa (Fig. 6)	2,754,490	0.288	12.661	42.243
Hong (Fig. 7)	3,117,913	0.196	14.027	71.566

large-scale data visualization, the rendering speed is more important. This paper shows a new way of efficiently rendering large-scale rendering data, through the DRM packets. It shows reasonable speed-ups.

Modern graphics systems use the programmable graphics pipelines. They execute the GPU instructions compiled from the shading language programs. For massively large-scale point clouds, the general purpose rendering pipelines are somewhat heavy to be processed. Our work is another way of processing the massively large-scale point clouds, with the special-purpose rendering pipeline. Our result shows this new approach works efficiently.

In the near future, we will release a customized application library for rendering massively large-scale point clouds. Especially the LiDAR point data will be processed most efficiently. We can extend the use of these special-purpose rendering library for various kinds of point-based data.

Acknowledgements This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (Grant 2019R1I1A3A01061310).

References

1. Baek N (2017) A fixed-function rendering pipeline with direct rendering manager support. In: ICITCS '17, pp 106–109
2. Baek N (2019) An emulation scheme for OpenGL SC 2.0 over OpenGL. J Supercomput 1:1–10
3. Baek N, Kim K (2019) Design and implementation of OpenGL SC 2.0 rendering pipeline. Clust Comput 1:1–6
4. Baek N, Ryu K (2015) Emulating OpenGL ES 2.0 over the desktop OpenGL. Clust Comput 18:165–175
5. Baek N, Shin W, Kim KJ (2017) Geometric primitive extraction from LiDAR-scanned point clouds. Clust Comput 20(1):741–748
6. Bailey MJ Mike bailey's vulkan page. http://cs.oregonstate.edu/~mjb/vulkan. Retrieved 17 Dec 2019
7. Benner WR Jr (2016) Laser scanners: technologies and applications. Amazon Digital Services
8. Cairo Graphics. http://www.cairographics.org/. Retrieved 17 Dec 2019
9. Dowman I (2004) Integration of LIDAR and IFSAR for mapping. Int Arch Photogramm Remote Sens 35:90–100
10. Faith RE (2016) The direct rendering manager: Kernel support for the direct rendering infrastructure. http://dri.sourceforge.net/doc/drm_low_level.html. Retrieved 17 Dec 2019
11. Fonseca J (2005) Direct rendering infrastructure: architecture. https://pdfs.semanticscholar.org/2ac2/20302febc42da62babf9d555717d65881ae8.pdf. Retrieved 17 Dec 2019
12. Gobbetti E, Marton F (2004) Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. Comput Gr 28(6):815–826
13. Group TKVW (2018) Vulkan 11.8.5–a specification. Khronos Group, Beaverton
14. Guo H (2014) Modern mathematics and applications in computer graphics and vision. World Scientific Pub, Singapore
15. Hermosilla P et al (2018) Monte carlo convolution for learning on non-uniformly sampled point clouds. ACM Trans Gr 37(6):235
16. Hughes JF et al (2013) Computer graphics: principles and practice, 3rd edn. Addison-Wesley, Reading
17. Kernel Mode Setting. https://wiki.archlinux.org/index.php/kernel_mode_setting. Retrieved 17 Dec 2019
18. Kessenich J (2016) The OpenGL shading language, language version: 4.50. Khronos Group, Beaverton
19. Lazar G (2017) Mastering Qt 5. Packt Publishing, Birmingham
20. Levoy M (1989) Design of a real-time high-quality volume rendering workstation. In: Proceedings VVS '89, pp 85–92
21. Luna F (2012) Introduction to 3D Game Programming with DirectX 11. Mercury Learning and Information
22. Malizia A (2006) Mobile 3D graphics. Springer, Berlin
23. Maloney RJ (2018) Low level X window programming: an introduction by examples. Springer, Berlin
24. Maltamo M, Næsset E (2016) Forestry applications of Airborne laser scanning: concepts and case studies. Springer, Berlin
25. Marshall GE, Stutz GE (2011) Handbook of optical and laser scanning, 2nd edn. CRC Press, Boca Raton
26. Mueller J, et al. (2018) Shading atlas streaming. In: Siggraph Asia '18
27. Munshi A, Leech J (2010) OpenGL ES common profile specification, version 2.0.25 (Full Specification). Khronos Group, Beaverton
28. Packard K, Anholt E (2008) The graphics execution manager: part of the direct rendering manager. https://lwn.net/Articles/283798/. Retrieved 17 Dec 2019
29. Pulli K et al (2007) Mobile 3D graphics: with OpenGL ES and M3G. Morgan Kaufmann Publishers Inc., Massachusetts
30. Segal M, Akeley K (2016) The OpenGL graphics system: a specification, version 4.5 (Core Profile). Khronos Group, Beaverton
31. Shan J, Toth CK (2018) Topographic laser ranging and scanning: principles and processing, 2nd edn. CRC Press, Boca Raton

32. Shin W, Baek N (2016) Editing LiDAR-based terrains with height and texture maps. In: ICISS ’16
33. Simpson RJ (2013) The OpenGL ES shading language, language version: 1.00. Khronos Group, Beaverton
34. Systems A (1993) Programming the display postscript system with X (APL). Addison-Wesley, Reading
35. Vosselman G, Mass HG (2010) Airborne and terrestrial laser scanning. CRC Press, Reading
36. Wernecke J (1994) The inventor mentor: programming object-oriented 3D graphics with open inventor. Addison-Wesley, Reading
37. Wimmer M, Scheiblauer C (2006) Instant points. In: Proceedings of Symposium on Point-Based Graphics ’06, pp 129–136
38. Young D (1994) The X window system: programming and applications with Xt, OSF/Motif, 2nd edn. Prentice Hall, Englewood Cliffs

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.